Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Not so easy, as it requires knowing what might possibly happen in the future

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Not so easy, as it requires knowing what might possibly happen in the future

An unsafe request will not be granted by the OS

The Banker's Algorithm

This algorithm, proposed by Dijkstra, is an example of how to do deadlock avoidance

The Banker's Algorithm

This algorithm, proposed by Dijkstra, is an example of how to do deadlock avoidance

The name comes from the concept of a bank lending money and then having it repaid

The Banker's Algorithm

This algorithm, proposed by Dijkstra, is an example of how to do deadlock avoidance

The name comes from the concept of a bank lending money and then having it repaid

It makes two tests

The Banker's Algorithm

This algorithm, proposed by Dijkstra, is an example of how to do deadlock avoidance

The name comes from the concept of a bank lending money and then having it repaid

It makes two tests

1. Feasibility test. To see if a request is possible

The Banker's Algorithm

This algorithm, proposed by Dijkstra, is an example of how to do deadlock avoidance

The name comes from the concept of a bank lending money and then having it repaid

It makes two tests

1. Feasibility test. To see if a request is possible
2. Safety test. To see if a request is safe (cannot lead to deadlock)

The Banker's Algorithm

Only requests that are both feasible and safe are granted

The Banker's Algorithm

Only requests that are both feasible and safe are granted

Note that a request that is feasible but not safe implies a resource is lying idle

The Banker's Algorithm

Only requests that are both feasible and safe are granted

Note that a request that is feasible but not safe implies a resource is lying idle

But we are erring on the side of safety at the cost of efficiency

The Banker's Algorithm

A request is *feasible* if, after granting the request, the total of allocated resource does not exceed the actual resource

The Banker's Algorithm

A request is *feasible* if, after granting the request, the total of allocated resource does not exceed the actual resource

That is, if we can actually satisfy the request. Don't allocate 10GB of memory if you only have 2GB

The Banker's Algorithm

A request is *feasible* if, after granting the request, the total of allocated resource does not exceed the actual resource

That is, if we can actually satisfy the request. Don't allocate 10GB of memory if you only have 2GB

Sometimes it can be all-or-nothing: allocate access to the sound card, or not

The Banker's Algorithm

A state is *safe* if there is *at least one* possible future sequence of resource allocations and releases by which all processes can complete their computation (never deadlock)

The Banker's Algorithm

A state is *safe* if there is *at least one* possible future sequence of resource allocations and releases by which all processes can complete their computation (never deadlock)

So make sure there is always an escape route of allocations and releases

The Banker's Algorithm

A state is *safe* if there is *at least one* possible future sequence of resource allocations and releases by which all processes can complete their computation (never deadlock)

So make sure there is always an escape route of allocations and releases

More than one route is better, but make sure there is at least one

The Banker's Algorithm

A state is *safe* if there is *at least one* possible future sequence of resource allocations and releases by which all processes can complete their computation (never deadlock)

So make sure there is always an escape route of allocations and releases

More than one route is better, but make sure there is at least one

A request is *safe* if, after granting the request, this leads to a safe state

The Banker's Algorithm

Dijkstra's Banking Algorithm:

> *Grant an allocation request only if this leads to a safe state*

The Banker's Algorithm

Dijkstra's Banking Algorithm:

> *Grant an allocation request only if this leads to a safe state*

This will *ensure* we are always deadlock-free, but can sometimes deny an allocation that might have been OK: it might have caused a deadlock, but by chance didn't happen to do so on some particular occasion

The Banker's Algorithm

In the implementation of this algorithm, for each process we need to know

The Banker's Algorithm

In the implementation of this algorithm, for each process we need to know

- The current allocation to that process

The Banker's Algorithm

In the implementation of this algorithm, for each process we need to know

- The current allocation to that process
- The maximum allocation that process might ever want

The Banker's Algorithm

Example. There are 12GB of memory and three processes sharing it

The Banker's Algorithm

Example. There are 12GB of memory and three processes sharing it

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	5	8
Available	2	

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	5	8
Available	2	

This is a safe state because all three processes can finish: we can demonstrate a path to completion for all processes

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	5	8
Available	2	

Process 2 currently has 4GB, but might eventually need 6

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	6	6
Process 3	5	8
Available	0	

If the 2GB available are given to Process 2

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	0	6
Process 3	5	8
Available	6	

Process 2 can finish releasing 6GB

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	4	4
Process 3	5	8
Available	3	

Then 3GB can be given to Process 1

The Banker's Algorithm

	Current allocation	Maximum need
Process 3	5	8
Available	7	

which can then finish, releasing 4GB

The Banker's Algorithm

	Current allocation	Maximum need
Process 3	8	8
Available	4	

And then 3GB can be given to Process 3

Deadlock

Avoidance

The Banker's Algorithm

	Current allocation	Maximum need
Available	12	

Which can now finish

The Banker's Algorithm

Thus there exists *a* path to completion for all processes where every process gets all the resources it might need: this is what the Banker's algorithm requires

The Banker's Algorithm

Thus there exists *a* path to completion for all processes where every process gets all the resources it might need: this is what the Banker's algorithm requires

This path may or may not be the actual one taken, e.g., Process 3 *might* exit without requiring that extra 3GB; this, of course, leads to another safe state

The Banker's Algorithm

Thus there exists *a* path to completion for all processes where every process gets all the resources it might need: this is what the Banker's algorithm requires

This path may or may not be the actual one taken, e.g., Process 3 *might* exit without requiring that extra 3GB; this, of course, leads to another safe state

But we still need to be careful with allocations, as it is possible to move from a safe state to an unsafe one

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	5	8
Available	2	

If Process 3 requests 1GB and this is granted

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	6	8
Available	1	

This is an unsafe state

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	6	8
Available	1	

Not necessarily deadlocked, but now we can't guarantee completion

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	6	8
Available	1	

No process can be guaranteed to get enough resources to complete

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	6	8
Available	1	

If we are lucky, a process might be able to finish without its maximum possible need

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	6	8
Available	1	

But an OS can't rely on luck

The Banker's Algorithm

There are several problems with this algorithm

The Banker's Algorithm

There are several problems with this algorithm

- There must be a fixed number of resources to allocate: a fair assumption, but not always true

The Banker's Algorithm

There are several problems with this algorithm

- There must be a fixed number of resources to allocate: a fair assumption, but not always true
- The population of processes must remain fixed: not in a general-purpose OS

The Banker's Algorithm

There are several problems with this algorithm

- There must be a fixed number of resources to allocate: a fair assumption, but not always true
- The population of processes must remain fixed: not in a general-purpose OS
- Processes must know their maximum needs in advance: very unlikely

The Banker's Algorithm

There are several problems with this algorithm

- There must be a fixed number of resources to allocate: a fair assumption, but not always true
- The population of processes must remain fixed: not in a general-purpose OS
- Processes must know their maximum needs in advance: very unlikely
- Safety detection is quite expensive to compute, particularly with multiple resources

The Banker's Algorithm

There are several problems with this algorithm

- There must be a fixed number of resources to allocate: a fair assumption, but not always true
- The population of processes must remain fixed: not in a general-purpose OS
- Processes must know their maximum needs in advance: very unlikely
- Safety detection is quite expensive to compute, particularly with multiple resources
- It can sometimes refuse a request that could have turned out to be OK (by luck, perhaps): this leads to idle resources